Rust Book - 4.1: Ownership

#rust Ownership

Ownership is Rust's most unique feature, and it enables Rust to make memory safety guarantees without needing a garbage collector.

Rust's **central feature ownership**.... it has deep implications for the rest of the language.

All programs have to manage the way they use a computer's memory while running. Some languages have garbage collection that constantly looks for no longer used memory as the program runs; in other languages, the programmer must explicitly allocate and free the memory.

memory is managed through a system of ownership with a set of rules that the compiler checks at compile time

Rust uses a third approach: memory is managed through a system of ownership with a set of rules that the compiler checks at compile time. None of the ownership features slow down your program while it's running.

The Stack and the Heap

The Stack and the Heap

In many programming languages, you don't have to think about the stack and the heap very often. But in a systems programming language like Rust, whether a value is on the stack or the heap has more of an effect on how the language behaves and why you have to make certain decisions.

Both the stack and the heap are parts of memory that are available to your code to use at runtime, but they are structured in different ways.

The **stack** stores values in the order it gets them and removes the values in the opposite order. FIFO Adding data is called*pushing onto the stack*, and removing data is called*popping off the stack*.

The stack is fast because of the way it accesses the data: it never has to search for a place to put new data or a place to get data from because that place is always the top. Another property that makes the stack fast is that all data on the stack must take up a known, fixed

size.

Data with a size unknown at compile time or a size that might change can be stored on the **heap** instead. The heap is less organized:

When you put data on the heap, you ask for some amount of space. The operating system finds an empty spot somewhere in the heap that is big enough, marks it as being in use, and returns a pointer, which is the address of that location. This process is called allocating on the heap, sometimes abbreviated as just "**allocating**."

Pushing values onto the stack is not considered allocating.

Because the pointer is a known, fixed size, you can store the pointer on the stack, but when you want the actual data, you have to follow the pointer.

Accessing data in the heap is slower than accessing data on the stack because you have to follow a pointer to get there.

Accessing data in the heap is slower than accessing data on the stack because you have to follow a pointer to get there.

Contemporary processors are faster if they jump around less in memory.

A processor can do its job better if it works on data that's close to other data (as it is on the stack) rather than farther away (as it can be on the heap).

Allocating a large amount of space on the heap can also take time.

When your code calls a function, the values passed into the function (including, potentially, pointers to data on the heap) and the function's local variables get pushed onto the stack. When the function is over, those values get popped off the stack.

Keeping track of what parts of code are using what data on the heap, minimizing the amount of duplicate data on the heap, and cleaning up unused data on the heap so you don't run out of space are all problems that ownership addresses.

Once you understand ownership, you won't need to think about the stack and the heap very often, but knowing that managing heap data is why ownership exists can help explain why it works the way it does.

Ownership Rules

Ownership Rules

- Each value in Rust has a variable that's called its **owner**.
- There can only be **one owner at a time**.
- When the owner goes **out of scope**, the value will be **dropped**.

A **variable** is valid from the point at which it's declared until the end of the current **scope**.

Memory and Allocation

However, the second part is different. With the String type, in order to support a mutable, growable piece of text, we need to allocate an amount of memory on the heap, unknown at compile time, to hold the contents.

This means:

- The memory must be requested from the operating system at runtime.
- We need a way of returning this memory to the operating system when we're done with ourString.

In languages with a **garbage collector (GC)** , the GC keeps track and cleans up memory that isn't being used anymore, and we don't need to think about it. Without a GC, it's our responsibility to identify when memory is no longer being used and call code to explicitly return it, just as we did to request it. Doing this correctly has historically been a difficult programming problem. If we forget, we'll waste memory. If we do it too early, we'll have an invalid variable. If we do it twice, that's a bug too.

… End up needing to pair exactly one allocate with exactly one free.

Rust takes a different path: the memory is automatically returned once the variable that owns it goes out of scope.

Rust takes a different path: the memory is automatically returned once the variable that owns it goes out of scope.

This pattern has a profound impact on the way Rust code is written. It may seem simple right now, but the behavior of code can be unexpected in more complicated situations

when we want to have multiple variables use the data we've allocated on the heap.

Ways Variables and Data Interact: Move

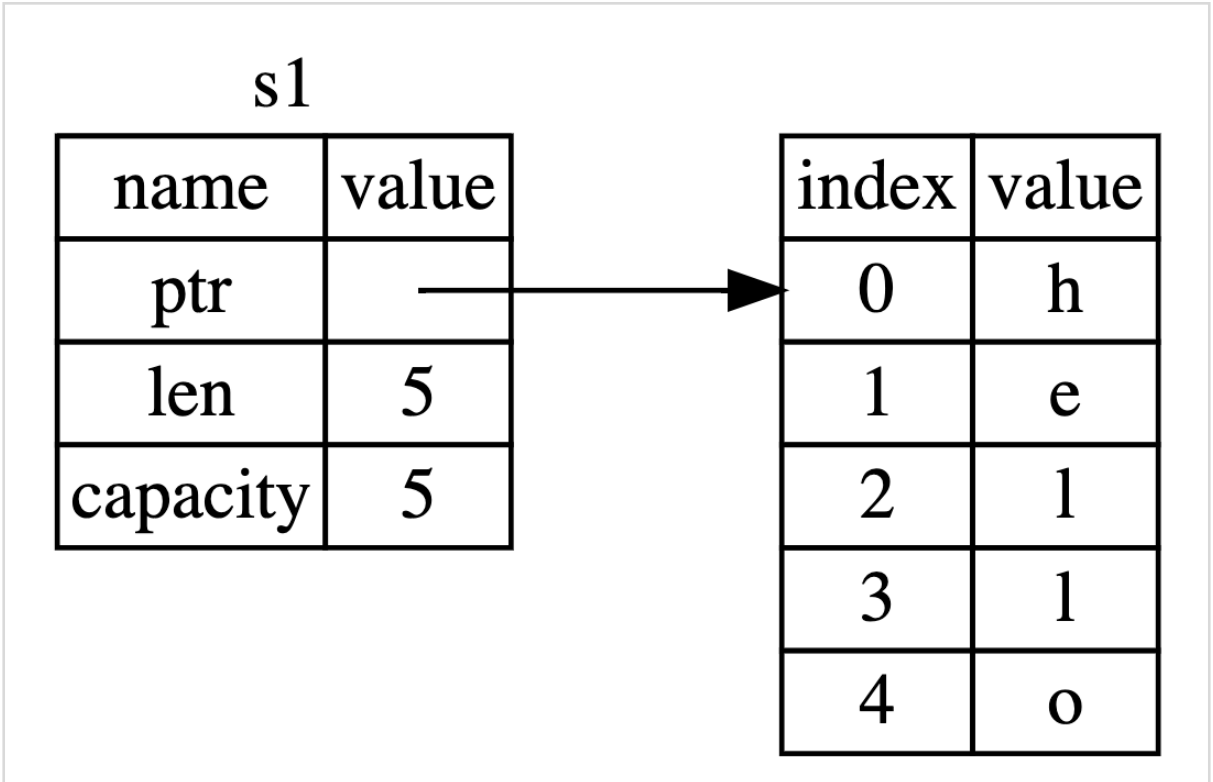
Multiple variables can interact with the same data in different ways

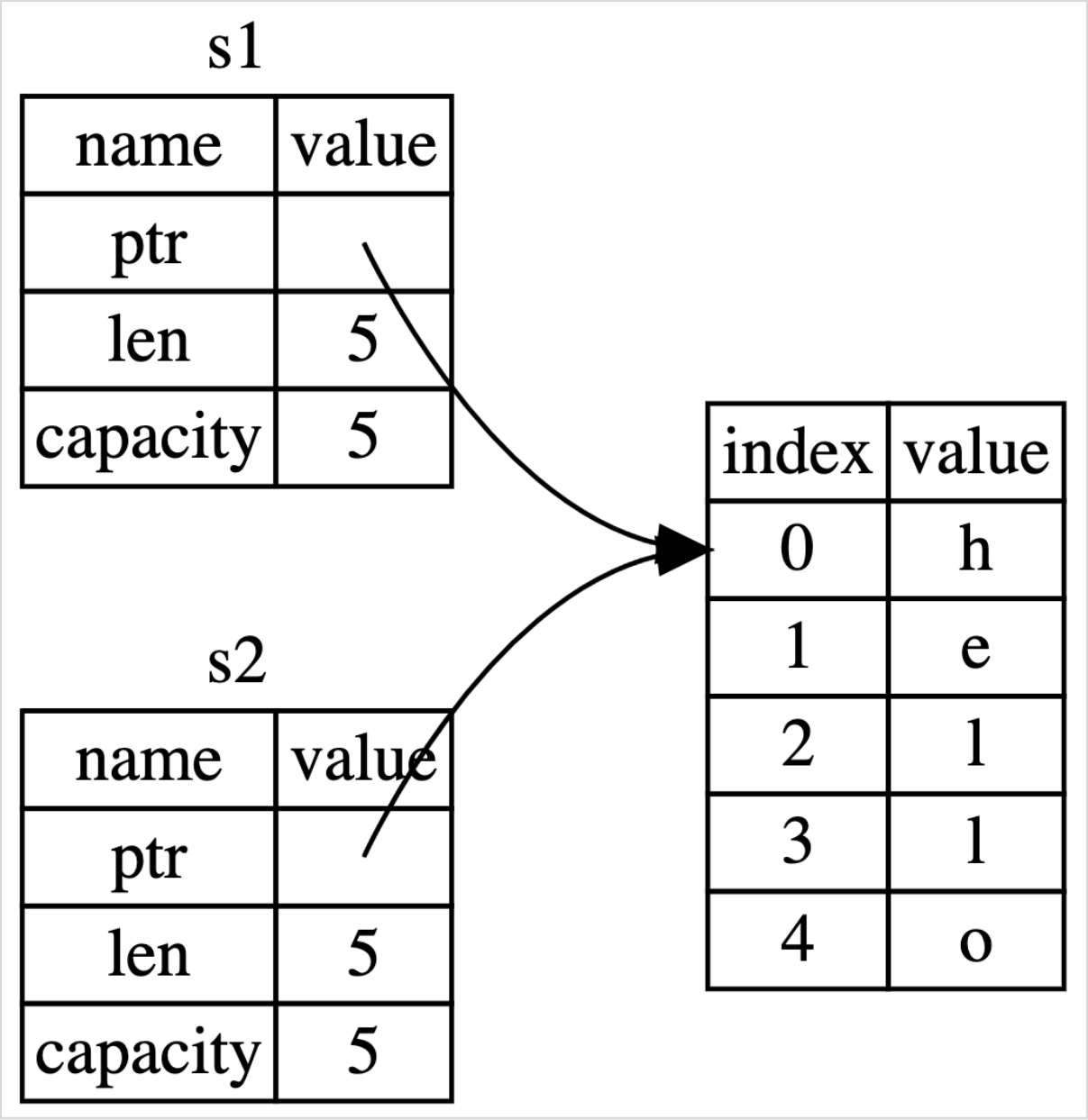
```
let x = 5;
let y = x;
```

```
let s1 = String::from("hello");
let s2 = s1;
```

A `String` is made up of three parts, shown on the left:

- a pointer to the memory that holds the contents of the string,
- a length, and
- a capacity.





When we assign `s1` to `s2`, the `String` data is copied, meaning we copy the pointer, the length, and the capacity that are on the stack. We do not copy the data on the heap that the pointer refers to.

Rust automatically calls the `drop` function and cleans up the heap memory for that variable. But Figure 4-2 shows both data pointers pointing to the same location. This is a problem: when `s2` and `s1` go out of scope, they will both try to free the same memory. This is known as a **double free error** and is one of the memory safety bugs we mentioned previously. Freeing memory twice can lead to memory corruption, which can potentially lead to security vulnerabilities.

To ensure memory safety, there's one more detail to what happens in this situation in Rust. Instead of trying to copy the allocated memory, Rust considers `s1` to no longer be valid

and, therefore, Rust doesn't need to free anything when `s1` goes out of scope. Check out what happens when you try to use `s1` after `s2` is created; it won't work:

```
let s1 = String::from("hello");
let s2 = s1;
println!("{}, world!", s1);
```

You'll get an error like this because Rust prevents you from using the invalidated reference:

```
error[E0382]: use of moved value: `s1`
 -> src/main.rs:5:28
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value used here after move
  |
  = note: move occurs because `s1` has type `std::string::String`, which does
  not implement the `Copy` trait
```

If you've heard the terms **shallow copy** and **deep copy** while working with other languages, the concept of copying the pointer, length, and capacity without copying the data probably sounds like making a shallow copy. But because Rust also invalidates the first variable, instead of being called a shallow copy, it's known as a **move**.

`s1` was *moved* into `s2`

In this example, we would say that `s1` was **moved** into `s2`.

With only `s2` valid, when it goes out of scope, it alone will free the memory, and we're done.

There's a design choice that's implied by this: Rust will never automatically create "deep" copies of your data. Therefore, any **automatic** copying can be assumed to be inexpensive in terms of runtime performance.

Ways Variables and Data Interact: Clone

If we **do** want to deeply copy the heap data of the `String`, not just the stack data, we can use a common method called `clone`.

```
let s1 = String::from("hello");
let s2 = s1.clone();
println!("s1 = {}, s2 = {}", s1, s2);
```

This works just fine and explicitly produces the behavior … where the heap data **does** get copied.

When you see a call to `clone`, you know that some arbitrary code is being executed and that code may be expensive. It's a visual indicator that something different is going on.

Stack-Only Data: Copy

```
let x = 5;
let y = x;
println!("x = {}, y = {}", x, y);
```

Types such as integers that have a known size at compile time are stored entirely on the stack, so copies of the actual values are quick to make.

That means there's no reason we would want to prevent `x` from being valid after we create the variable `y`. In other words, there's no difference between deep and shallow copying here, so calling `clone` wouldn't do anything different from the usual shallow copying and we can leave it out.

Rust has a special annotation called the `Copy` trait that we can place on types like integers that are stored on the stack. If a type has the `Copy` trait, an older variable is still usable after assignment. Rust won't let us annotate a type with the `Copy` trait if the type, or any of its parts, has implemented the `Drop` trait.

If the type needs something special to happen when the value goes out of scope and we add the `Copy` annotation to that type, we'll get a compile-time error.

As a general rule, any group of simple scalar values can be `Copy`, and nothing that requires allocation or is some form of resource is `Copy`.

Some of the types that are `Copy`:

- All the integer types, such as `u32`.
- The Boolean type, `bool`, with values `true` and `false`.
- All the floating point types, such as `f64`.
- The character type, `char`.
- Tuples, if they only contain types that are also `Copy`. e.g. `(i32, i32)` is `Copy`, but `(i32, String)` is not.

Ownership and Functions

Ownership and Functions

The semantics for passing a value to a function are similar to those for assigning a value to a variable. Passing a variable to a function will move or copy, just as assignment does.

```
fn main() {
    let s = String::from("hello");  // s comes into scope
    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here
    let x = 5;                      // x comes into scope
    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved,
nothing special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.
```

The ownership of a variable follows the same pattern every time: assigning a value to another variable moves it. When a variable that includes data on the heap goes out of scope, the value will be cleaned up by `drop` unless the data has been moved to be owned by another variable.

Taking ownership and then returning ownership with every function is a bit tedious. What if we want to let a function use a value but not take ownership? It's quite annoying that anything we pass in also needs to be passed back if we want to use it again, in addition to any data resulting from the body of the function that we might want to return as well.

It's possible to return multiple values using a tuple:

```
fn main() {
    let s1 = String::from("hello");
    let (s2, len) = calculate_length(s1);
    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String
    (s, length)
}
```

References and Borrowing

References and Borrowing

The issue with the tuple code is that we have to return the `String` to the calling function so we can still use the `String` after the call to `calculate_length`, because the `String` was moved into `calculate_length`.

Use a `calculate_length` function that has a reference to an object as a parameter instead of taking ownership of the value:

```
fn main() {
```

```
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

First, notice that all the tuple code in the variable declaration and the function return value is gone. Second, note that we pass `&s1` into `calculate_length` and, in its definition, we take `&String` rather than `String`.

These ampersands are **references**, and they allow you to refer to some value without taking ownership of it.

The opposite of referencing by using `&` is **dereferencing**, which is accomplished with the dereference operator, `*`.

The `&s1` syntax lets us create a reference that **refers** to the value of `s1` but does not own it. Because it does not own it, the value it points to will not be dropped when the reference goes out of scope. Likewise, the signature of the function uses `&` to indicate that the type of the parameter `s` is a reference.

The scope in which the variable `s` is valid is the same as any function parameter's scope, but we don't drop what the reference points to when it goes out of scope because we don't have ownership. When functions have references as parameters instead of the actual values, we won't need to return the values in order to give back ownership, because we never had ownership.

We call having references as function parameters **borrowing**.

```
fn main() {
    let s = String::from("hello");
    change(&s);
}
fn change(some_string: &String) {
```

```
    some_string.push_str(“, world”);
}
```

```
error[E0596]: cannot borrow immutable borrowed content `*some_string` as
mutable
 -> error.rs:8:5
  |
7 | fn change(some_string: &String) {
  |                        ---- use `&mut String` here to make mutable
8 |     some_string.push_str(“, world”);
  |     ^^^^^^^^^^^^ cannot borrow as mutable
```

Just as variables are immutable by default, so are references. We’re not allowed to modify something we have a reference to.

Mutable References

Mutable References

```
fn main() {
    let mut s = String::from(“hello”);
    change(&mut s);
}
fn change(some_string: &mut String) {
    some_string.push_str(“, world”);
}
```

Mutable references have one big restriction: you can have only one mutable reference to a particular piece of data in a particular scope. This code will fail:

```
let mut s = String::from(“hello”);
let r1 = &mut s;
let r2 = &mut s;
```

```
println!(“{}, {}”, r1, r2);
```

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
 –> src/main.rs:5:10
  |
4 | let r1 = &mut s;
  |          —— first mutable borrow occurs here
5 | let r2 = &mut s;
  |          ^^^^^^ second mutable borrow occurs here
6 | println!(“{}, {}”, r1, r2);
  |                    – borrow later used here
```

This restriction allows for mutation but in a very controlled fashion. It’s something that new Rustaceans struggle with, because most languages let you mutate whenever you’d like.

The benefit of having this restriction is that Rust can prevent data races at compile time. A **data race** is similar to a race condition and happens when these three behaviors occur:

- Two or more pointers access the same data at the same time.
- At least one of the pointers is being used to write to the data.
- There’s no mechanism being used to synchronize access to the data.

Data races cause undefined behavior and can be difficult to diagnose and fix when you’re trying to track them down at runtime; Rust prevents this problem from happening because it won’t even compile code with data races!
We can use curly brackets to create a new scope, allowing for multiple mutable references, just not simultaneous ones:

```
let mut s = String::from(“hello”);
{
    let r1 = &mut s;
} // r1 goes out of scope here, so we can make a new reference with no
problems.
let r2 = &mut s;
```

A similar rule exists for combining mutable and immutable references.

```
let mut s = String::from("hello");
let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM
println!("{}, {}, and {}", r1, r2, r3);
```

Error:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
 -> src/main.rs:6:10
  |
4 | let r1 = &s; // no problem
  |          - immutable borrow occurs here
5 | let r2 = &s; // no problem
6 | let r3 = &mut s; // BIG PROBLEM
  |          ^^^^^^ mutable borrow occurs here
7 |
8 | println!("{}, {}, and {}", r1, r2, r3);
  |                            -- borrow later used here
```

We **also** cannot have a mutable reference while we have an immutable one. Users of an immutable reference don't expect the values to suddenly change out from under them! However, multiple immutable references are okay because no one who is just reading the data has the ability to affect anyone else's reading of the data.

Dangling References

Dangling References

In languages with pointers, it's easy to erroneously create a **dangling pointer** , a pointer that references a location in memory that may have been given to someone else, by freeing some memory while preserving a pointer to that memory. In Rust, by contrast, the compiler guarantees that references will never be dangling references: if you have a reference to some data, the compiler will ensure that the data will not go out of scope before the reference to the data does.

```
fn main() {
    let reference_to_nothing = dangle();
}
fn dangle() -> &String {
    let s = String::from("hello");
    &s
}
```

Error:

```
error[E0106]: missing lifetime specifier
 --> main.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is
no value for it to be borrowed from
```

Because `s` is created inside `dangle`, when the code of `dangle` is finished, `s` will be deallocated. But we tried to return a reference to it. That means this reference would be pointing to an invalid `String`. Rust won't let us do this.

```
fn no_dangle() -> String {
    let s = String::from("hello");
    s
}
```

Ownership is moved out, and nothing is deallocated.

The Rules of References

* At any given time, you can have **either** one mutable reference **or** any number of immutable references.
* References must always be valid.

String Slices

String Slices

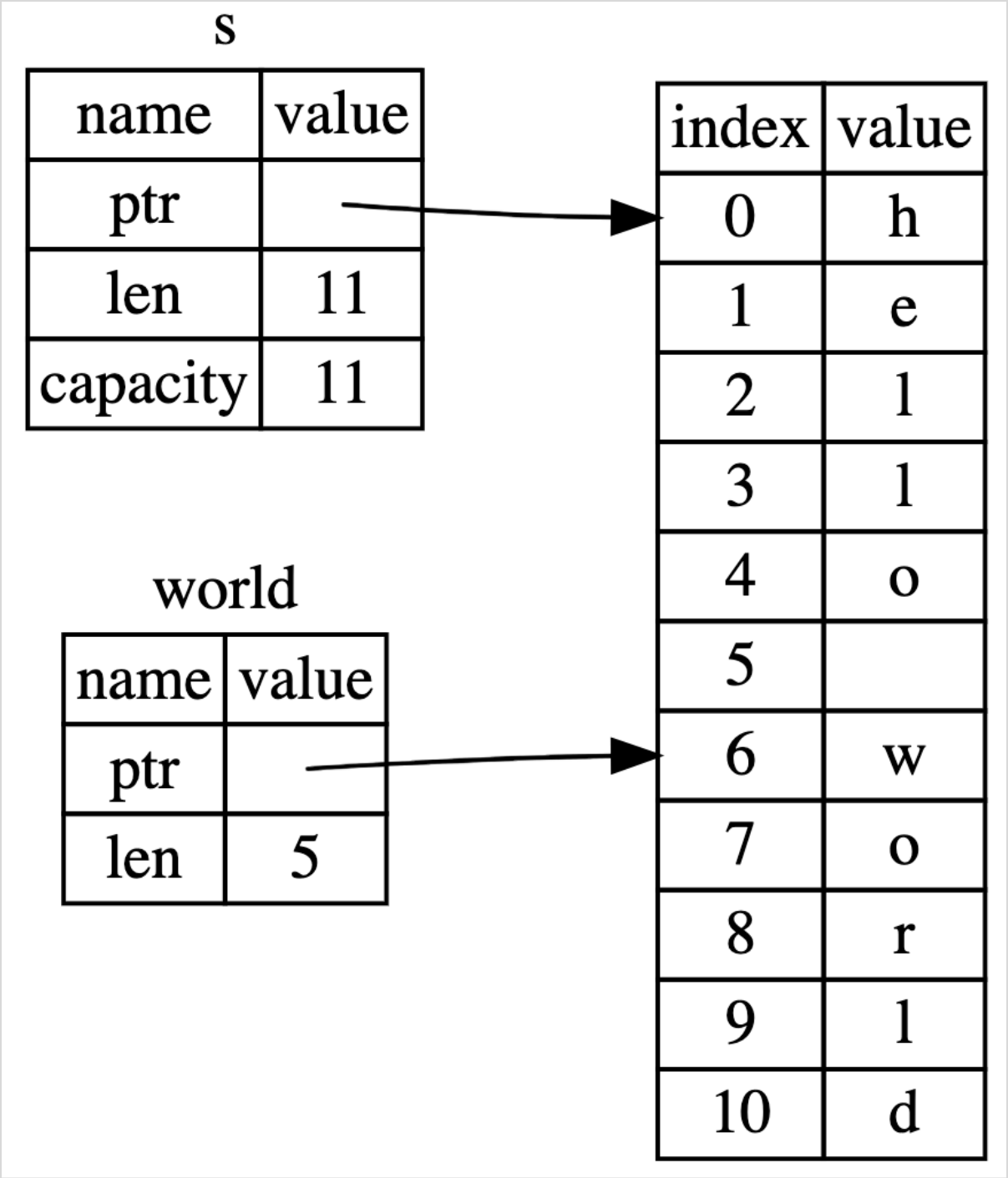
A **string slice** is a reference to part of a `String`:

```
let s = String::from("hello world");
let hello = &s[0..5];
let world = &s[6..11];
```

Rather than a reference to the entire `String`, it's a reference to a portion of the `String`. The `start..end` syntax is a range that begins at `start` and continues up to, but not including, `end`. If we wanted to include `end`, we can use `..=` instead of `..`:

The `=` means that we're including the last number, if that helps you remember the difference between `..` and `..=`.

We can create slices using a range within brackets by specifying `[starting_index..ending_index]`, where `starting_index` is the first position in the slice and `ending_index` is one more than the last position in the slice. Internally, the slice data structure stores the starting position and the length of the slice, which corresponds to `ending_index` minus `starting_index`. So in the case of `let world = &s[6..11];`, `world` would be a slice that contains a pointer to the 7th byte of `s` with a length value of 5.




```
let len = s.len();
let slice = &s[0..len];
let slice = &s[..];
```

The type that signifies "**string slice**" is written as `&str`:

```
fn first_word(s: &String) -> &str {
```

```
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }
    &s[..]
}
```

Old version:

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }
    s.len()
}
```

```
fn main() {
    let mut s = String::from("hello world");
    let word = first_word(&s);
    s.clear(); // error!
    println!("the first word is: {}", word);
}
```

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
  --> src/main.rs:10:5
   |
8  |     let word = first_word(&s);
   |                           -- immutable borrow occurs here
```

```
9  |
10 |     s.clear(); // error!
   |     ^^^^^^^^^ mutable borrow occurs here
11 |
12 |     println!("the first word is: {}", word);
   |                                       ---- borrow later used here
```

Recall from the borrowing rules that if we have an immutable reference to something, we cannot also take a mutable reference. Because `clear` needs to truncate the `String`, it tries to take a mutable reference, which fails.

String Literals Are Slices

```
let s = “Hello, world!”;
```

The type of `s` here is `&str`: it’s a slice pointing to that specific point of the binary. This is also why string literals are immutable; `&str` is an immutable reference.

String Slices as Parameters

Knowing that you can take slices of literals and `String` values leads us to one more improvement on `first_word`, and that’s its signature:

```
fn first_word(s: &String) -> &str {
```

A more experienced Rustacean would write this signature because it allows us to use the same function on both `String` values and `&str` values.

```
fn first_word(s: &str) -> &str {
```

Defining a function to take a string slice instead of a reference to a `String` makes our API more general and useful without losing any functionality:

```
fn main() {
    let my_string = String::from("hello world");
    // first_word works on slices of `String`s
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";
    // first_word works on slices of string literals
    let word = first_word(&my_string_literal[..]);

    // Because string literals *are* string slices already,
    // this works too, without the slice syntax!
    let word = first_word(my_string_literal);
}
```

Other Slices

Other Slices

```
let a = [1, 2, 3, 4, 5];
let slice = &a[1..3];
```

This slice has the type `&[i32]`. It works the same way as string slices do, by storing a reference to the first element and a length.

Summary

The concepts of ownership, borrowing, and slices ensure memory safety in Rust programs at compile time. The Rust language gives you control over your memory usage in the same way as other systems programming languages, but having the owner of data automatically clean up that data when the owner goes out of scope means you don't have to write and debug extra code to get this control.